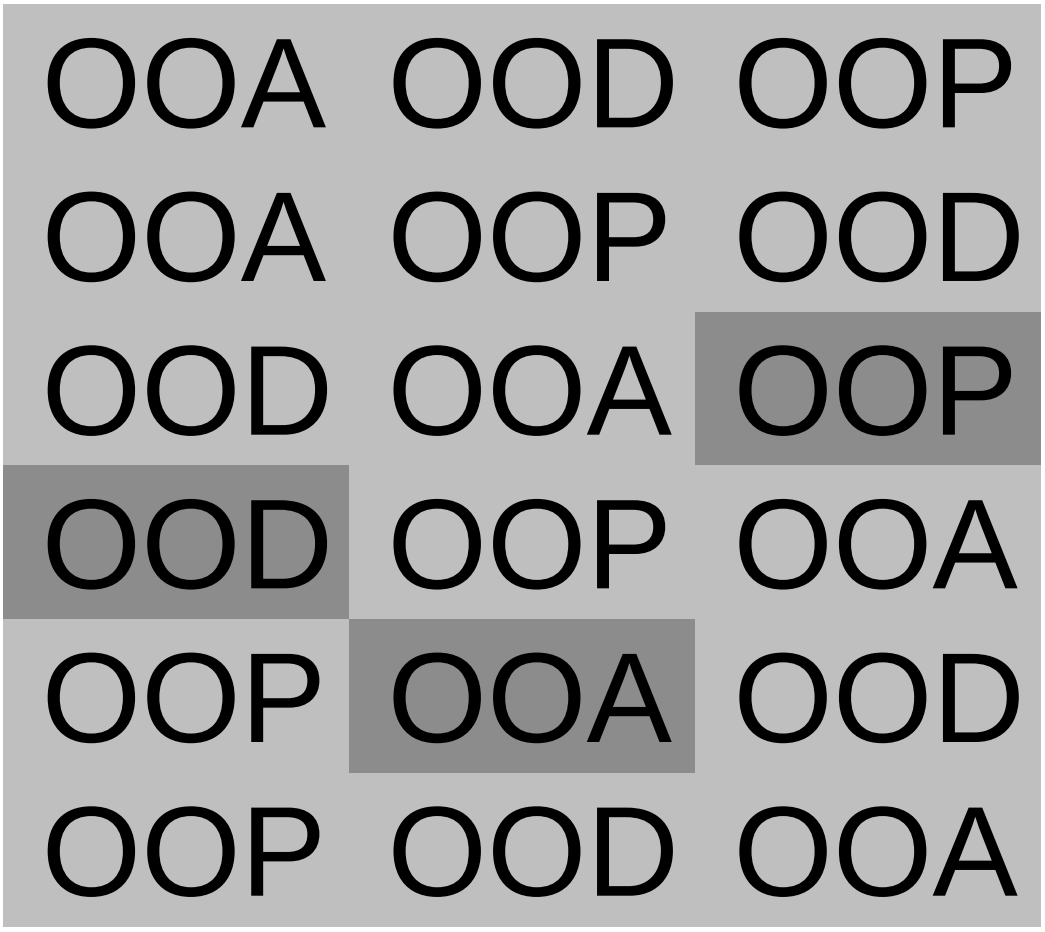
Java Springframework
Übungen



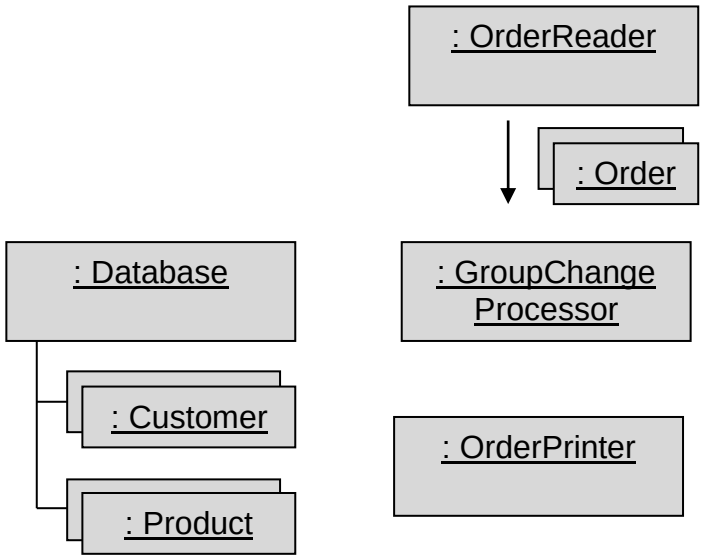
Johannes Nowak

Inhalt

1 Start

Das Start-Projekt, das im folgenden beschrieben wird, heißt `z01-Start` (und ist unter diesem Namen im Workspace enthalten).

Zunächst ein Schaubild:



Bei der zu bearbeitenden Applikation handelt es sich um ein Gruppenwechsel-Programm. Sie verarbeitet die Einträge einer CSV-Datei zu einer Druckliste. Die CSV-Datei enthält Aufträge. Jede Auftragszeile enthält eine Kundennummer, eine Produktnummer und eine Bestellmenge:

orders.txt

```
1000,100,10
1000,200,10
1000,300,20
2000,200,50
3000,100,10
3000,300,20
```

Die Eingabe ist gruppenförmig strukturiert (zunächst alle Aufträge für den Kunden 1000, dann alle für den Kunden 2000 und schließlich alle für den Kunden 3000).

Aufgrund dieser Eingabe wird folgende Druckliste erzeugt:

```
Orders
   1000 Meier
        100 Veltins      10    11     110
```

```
        200 Bitburger    10    22     220
        300 Krombacher   20    33     660
                                     -----
                                       990
   2000 Mueller
        200 Bitburger    50    22    1100
                                     -----
                                      1100
   3000 Schulte
        100 Veltins      10    11     110
        300 Krombacher   20    33     660
                                     -----
                                       770
                                     =====
                                      2860
```

Um diese Druckliste zu erzeugen, benötigen wir natürlich Referenz-Daten: die Namen der Kunden und die Namen und die Einzelpreise der Produkte.

Diese Daten liegen in festverdrahtetet Form vor – in der Klasse `Database`:

appl.Database

```
package appl;
// ...
public class Database {

    private final Map<Integer,Customer> customers =
        new HashMap<Integer,Customer>();

    private final Map<Integer,Product> products =
        new HashMap<Integer,Product>();

    public Database() {
        this.customers.put(1000, new Customer(1000, "Meier"));
        this.customers.put(2000, new Customer(2000, "Mueller"));
        this.customers.put(3000, new Customer(3000, "Schulte"));
        this.products.put(100, new Product(100, "Veltins", 11));
        this.products.put(200, new Product(200, "Bitburger", 22));
        this.products.put(300, new Product(300, "Krombacher", 33));
    }

    public Customer findCustomer(int number) {
        return this.customers.get(number);
    }
    public Product findProduct(int number) {
        return this.products.get(number);
    }
}
```

Customer und Product sind einfache POJO-Klassen:

appl.Customer

```
package appl;

public class Customer {
    private int number;
    private String name;
    public Customer(int number, String name) {
        this.number = number;
        this.name = name;
    }
    // getter, setter...
}
```

appl.Product

```
package appl;

public class Product {
    private int number;
    private String name;
    private int price;
    public Product(int number, String name, int price) {
        this.number = number;
        this.name = name;
        this.price = price;
    }
    // getter, setter...
}
```

Die Eingabezeilen werden transformiert in `Order`-Objekte:

appl.Order

```
package appl;

public class Order {

    private final int customerNumber;
    private final int productNumber;
    private final int amount;

    public Order(int customerNumber, int productNumber, int amount) {
        this.customerNumber = customerNumber;
        this.productNumber = productNumber;
        this.amount = amount;
    }

    // getter
}
```

Für das Einlesen der `Order`s ist die Klasse OrderReader zuständig:

appl.OrderReader

```
package appl;
// ...
public class OrderReader {
    private BufferedReader reader;

    public OrderReader(String filename) {
        try {
            this.reader = new BufferedReader(
                new InputStreamReader(new FileInputStream(filename)));
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public void close() {
        if (reader == null)
            throw new RuntimeException("reader not open");
        try {
            reader.close();
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public Order read() {
        try {
            for(String line = reader.readLine();
                    line != null;
                    line = reader.readLine()) {
                line = line.trim();
                if (line.isEmpty())
                    continue;
                String[] tokens = line.split(",");
                int customerNumber = Integer.parseInt(tokens[0].trim());
                int productNumber = Integer.parseInt(tokens[1].trim());
                int amount = Integer.parseInt(tokens[2].trim());
                return new Order(customerNumber, productNumber, amount);
            }
            return null;
        }
        catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Die `read`-Methode liefert jeweils das nächste `Order`-Objekt – oder `null`, wenn das Ende der Eingabe erreicht ist.

Für die Produktion der Ausgaben ist ein `OrderPrinter` zuständig:

appl.OrderPrinter

```
package appl;

public class OrderPrinter {
    public void printBegin() { ... }
    public void printGroupBegin(Customer customer) { ... }
    public void printItem(Order order, Product product, int value) { ... }
    public void printGroupEnd(int groupSum) { ... }
    public void printEnd(int totalSum) { ... }
}
```

Der eigentliche Gruppenwechsel-Algorithmus ist in der `run`-Methode der Klasse `GroupChangeProcessor` implementiert:

appl.GroupChangeProcessor

```
package appl;

public class GroupChangeProcessor {
    public void run() {
        final OrderReader reader = new OrderReader("src/orders.txt");
        final Database database = new Database();
        final OrderPrinter printer = new OrderPrinter();
        Order order = reader.read();
        printer.printBegin();
        int totalSum = 0;
        while(order != null) {
            int groupSum = 0;
            final int customerNumber = order.getCustomerNumber();
            final Customer customer = database.findCustomer(customerNumber);
            printer.printGroupBegin(customer);
            while(order != null &&
                    order.getCustomerNumber() == customerNumber) {
                final Product product =
                    database.findProduct(order.getProductNumber());
                final int itemValue = order.getAmount() * product.getPrice();
                printer.printItem(order, product, itemValue);
                groupSum += itemValue;
                order = reader.read();
            }
            printer.printGroupEnd(groupSum);
            totalSum += groupSum;
        }
        printer.printEnd(totalSum);
        reader.close();
    }
}
```

Das Hauptprogramm schließlich sieht wir folgt aus:

appl.Application

```
package appl;

public class Application {
    public static void main(String[] args) {
        new GroupChangeProcessor().run();
    }
}
```

2 Einführung von Interfaces

Kopieren Sie das Start-Projekt nach `y02-Interfaces`.

Spezifizieren Sie die Klassen `OrderReader`, `Database` und `OrderPrinter` über Interfaces.

Die Interfaces sollen genauso heißen wie die bisherigen Klassen; die Implementierung solle heißen `CsvOrderReader`, `DatabaseDummy` und `SimpleOrderPrinter`.

Ändern Sie dann die Klasse `GroupChangeProcessor` dergestalt, das die `main`-Methode wie folgt ausschaut:

```
    public static void main(String[] args) {
        final OrderReader reader = new CsvOrderReader("src/orders.txt");
        final Database database = new DatabaseDummy();
        final OrderPrinter printer = new SimpleOrderPrinter();
        new GroupChangeProcessorImpl(reader, database, printer).run();
    }
```

3 XMLContext-ConstructorInjection

Kopieren Sie das letzte Projekt nach `y03-XMLContext-ConstructorInjection`.

Lassen Sie die an der Applikation beteiligten Objekte von Spring erzeugen.

Schreiben Sie eine `spring.xml`, mittels derer zunächst der `CsvOrderReader`, der `DatabaseDummy` und der `SimpleOrderPrinter` erzeugt wird (wobei jedes Objekt mit einer `id` versehen wird).

Dann soll der `GroupChangeProcessor` erzeugt werden, wobei die drei erzeugten Hilfsobjekte dem Konstruktor der `GroupChangeProcessor`-Klasse übergeben wird (Constructor-Injection).

Der Name der Eingabedatei sollte in der `spring.xml` hinterlegt sein (und dem `CsvOrderReader` übergeben werden).

Die `main`-Methode sollte dann wie folgt ausschauen:

```
    public static void main(String[] args) {
        try (ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("spring.xml")) {
            ctx.getBean(GroupChangeProcessor.class).run();
        }
    }
```

4 XMLContext-PropertyInjection

Kopieren Sie das letzte Projekt nach `y04-XMLContext-PropertyInjection`.

Bauen Sie die Anwendung derart um, dass statt Constructor-Injection nun Property-Injection genutzt wird.

Die `main`-Methode wir dabei unverändert bleiben.

5 XMLContext-PropertyFile

Kopieren Sie das letzte Projekt nach `y05-XMLContext-PropertiyFile`

Der Name der Eingabedatei war bislang in der `spring.xml` hinterlegt.

Dieser Name soll nun in einer Property-Datei hinterlegt werden:

orders.properties

```
filename = src/orders.txt
```

Benutzen Sie das `PropertyPlaceholderConfigurer`-Konzept, um den Dateinamen aus dieser Properties-Datei auszulesen und diesen Namen an den `CsvOrderReader` zu übergeben.

6 XMLContext-Inline

Kopieren Sie das letzte Projekt nach `y06-XMLContext-Inline`

In der spring.xml waren alle Beans als "öffentlich" definiert (jede hatte eine eigene ID und war auf der äußersten Ebene des XML-Baumes definiert).

Definieren Sie die drei Helper-Beans (`CsvOrderReader`, `DatabaseDummy` und `SimpleOrderPrinter`) inline (also als namenlose Beans im Kontext der `GroupChangeProcessor`-Bean).

7 Annotations-FieldInjection

Kopieren Sie das letzte Projekt nach `y07-Annotations-FieldInjection`

Löschen Sie die `spring.xml`-Datei.

Annotieren Sie alle Services (und auch den `GroupChangeProcssor`!) mit `@Component`.

In der `CsvOrderReader`-Klasse benötigen Sie zusätzlich eine `@PropertySource`- und eine `@Value`-Annotation. Das, was der Konstruktor dieser Klasse bislang erledigte, wird eine `@PostConstruct`-Methode übernehmen müssen.

In der `GroupChangeProcessor`-Klasse sollten Sie Field-Injection benutzen (`@Autowired`).

Die `main`-Methode sollte wie folgt ausschauen:

```
    public static void main(String[] args) {
        try (AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext("appl")) {
            ctx.getBean(GroupChangeProcessor.class).run();
        }
    }
```

8 Annotations-ConstructorInjection

Kopieren Sie das letzte Projekt nach `y08-Annotations-ConstructorInjection`

Benutzen Sie statt Field-Injection nun Constructor-Injection!

Die `main`-Methode sollte unverändert bleiben.

9 Annotations-Configuration

Kopieren Sie das letzte Projekt nach `y09-Annotations-Configuration`

Entfernen Sie alle Spring-Annotationen (`@Component`, `@PropertySource`, `@Value`, `@Autowired`).

Erstellen Sie eine `ApplConfig`-Klasse, die mit `@Configuration` annotiert ist. In dieser Klasse sollen nun die Services erzeugt und verdrahtet werden. Zu diesem Zweck sollte sie vier mit `@Bean` annotierte Methoden enthalten.

Die `ApplConfig`-Klasse sollte nun `@PropertySource` und `@Value` enthalten – und der Dateiname sollte über einen Konstruktor an den `CsvOrderReader` weitergereicht werden.

Die `main`-Methode bleibt dabei unberührt.

10 AOP

Kopieren Sie das letzte Projekt nach `y10-AOP`

Die Zugriffe auf die Datenbank (`getCustomer`, `getProduct`) sollen getract werden.

Erstellen Sie zu diesem Zweck eine Klasse `TraceBeforeAfterAdvice`, welche die Interfaces `MethodBeforeAdvice`, `AfterReturningAdvice` und `ThrowsAdvice` implementiert.

Benutzen Sie in der `ApplConfig` eine `ProxyFactoryBean`, um den Trace-Aspect hinzuzufügen.

11 JDBC

Kopieren Sie das letzte Projekt nach `y11-JDBC`

Entfernen Sie zunächst wieder den in der letzten Übung hinzugefügten Trace-Aspekt.

Erstellen Sie eine "richtige" Datenbank mit folgenden Tabellen (`create.sql`):

```
create table customer (
      number integer,
      name varchar(64),
      primary key (number)
);
create table product (
      number integer,
      name varchar(64),
      price integer,
      primary key (number)
);
insert into customer values(1000, 'Meier');
insert into customer values(2000, 'Mueller');
insert into customer values(3000, 'Schulte');
insert into product values(100, 'Veltins', 11);
insert into product values(200, 'Bitburger', 11);
insert into product values(300, 'Krombacher', 11);
```

Implementieren Sie das Interface `Database` nun in einer Klasse `JdbcDatabase`. Leiten Sie diese Klasse von `JdbcDaoSupport` ab.

In dieser Klasse können Sie folgende `RowMapper` nutzen:

```
    private final RowMapper<Customer> customerMapper = (rs, rowNumber) ->
        new Customer(rs.getInt("number"),
                     rs.getString("name"));

    private final RowMapper<Product> productMapper = (rs, rowNumber) ->
        new Product(rs.getInt("number"),
                    rs.getString("name"),
                    rs.getInt("price"));
```

Bei der Implementierung der `getCustomer`- und der `getProduct`-Methoden sollten Sie die von `JdbcDaoSupport` geerbte Methode `getJdbcTeemplate` nutzen.

Die `ApplConfig`-Klasse sollte u.a. (!) um folgende Elemente erweitert werden:

```
@PropertySource("classpath:db.properties")
```

```
    @Value("${db.driver}")
    private String driver;
```

```
@Value("${db.url}")
private String url;

@Value("${db.user}")
private String user;

@Value("${db.password}")
private String password;

@Bean
public DataSource dataSource() {
    Log.log();
    return new SimpleDataSource(
        this.driver, this.url, this.user, this.password);
}
```

12 JPA

Kopieren Sie das letzte Projekt nach `y12-JPA`

Benutzen Sie nun eine `JpaDatabase`-Klasse, welche die Datenbank-Zugriffe mit den Mitteln von JPA implementiert.

Denken Sie daran, die `Customer` und die `Product`-Klassen mit JPA-Annotationen auszustatten.

Sie benötigen eine Datei `META-INF/persistence.xml`.

Und in der Klasse `JpaDatabase` definieren Sie einen `EntityManager`, der mittels `@PersistenceContext` injiziert wird.

Und schließlich benötigen Sie in der `ApplConfig` noch folgenden Eintrag:

```
    @Bean
    public LocalEntityManagerFactoryBean managerFactory() {
        return new LocalEntityManagerFactoryBean();
    }
```